# 1. 深度学习（一）

深度学习是一种机器学习的分支，它利用一种称为人工神经网络的模型来对数据进行建模和分析。与传统的机器学习算法不同，深度学习能够从大量的数据中自动学习模式，而无需显式编程指导。

在深度学习中，神经网络由多个层组成，每一层都包含多个节点或神经元。这些神经元通过连接进行交流，并使用一些数学函数来计算其输出。深度学习模型的训练过程通常涉及使用反向传播算法来调整网络的权重，以最小化模型的预测误差。

深度学习在计算机视觉、自然语言处理、语音识别等许多领域中都有广泛应用，例如图像分类、物体检测、机器翻译和语音识别等。深度学习算法的发展和应用对于人工智能的发展和推动具有重要意义。

## 1.1 多层感知机

多层感知机（Multilayer Perceptrons），简称 MLPs，是当前深度学习最基础的模型。

### 1.1.1 隐藏层(Hidden layer)

多层感知机在单层神经网络的基础上引入了一到多个隐藏层（hidden layer）。隐藏层位于输入层和输出层之间。图 1.1 展示了一个多层感知机的神经网络图，它含有两个隐藏层，每层中各有 4 个隐藏单元。

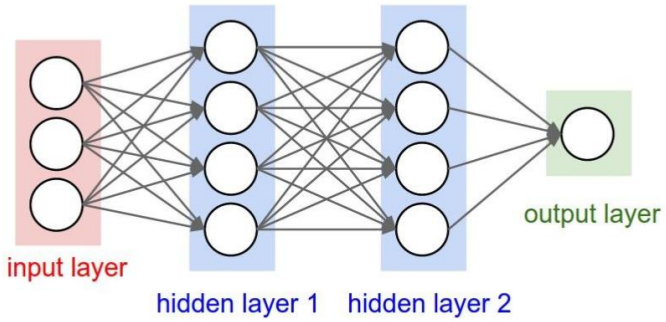



图 1.1 多层感知机

如图 1.1 所示，多层感知机中输入和输出个数分别为 3 和 1，中间的 2 个隐藏层中共包含了 8 个隐藏单元（hidden unit）。由于输入层不涉及计算，图 1.1 中的多层感知机的层数为 3。隐藏层中的神经元和输入层中各个输入完全连接，输出层中的神经元和隐藏层中的各个神经元也完全连接。因此，多层感知机中的隐藏层和输出层都是全连接层。

### 1.1.2 激活函数(Activate function)

激活函数是一种添加到人工神经网络中的函数，旨在帮助网络学习数据中的复杂和非线性的模式。激活函数定义了神经元在给定的输入或输入集合下的输出，决定了是否传递信号以及要发射给下一个神经元的内容。 激活函数可以分为饱和激活函数和非饱和激活函数，例如 Sigmoid， ReLU ，Tanh 等。

为什么使用激活函数？

因为神经网络中每一层的输入输出都是一个线性求和的过程，下一层的输出只是承接了上一层输入函数的线性变换，所以如果没有激活函数，那么无论你构造的神经网络多么复杂，有多少层，最后的输出都是输入的线性组合，纯粹的线性组合并不能够解决更为复杂的问题。而引入激活函数之后，我们会发现常见的激活函数都是非线性的，因此也会给神经元引入非线性元素，使得神经网络可以逼

近其他的任何非线性函数，这样可以使得神经网络应用到更多非线性模型中。

在神经网络中，神经元的工作原理如图 1.2 进行表示：

Take input and multiply by the neuron's weight.
Add bias*
Feed the result, x, to the activation function: f(x)
Take the output and transmit to the next layer of neurons.

图 1.2 神经元工作原理

上述过程的数学可视化过程如图 1.3 所示：

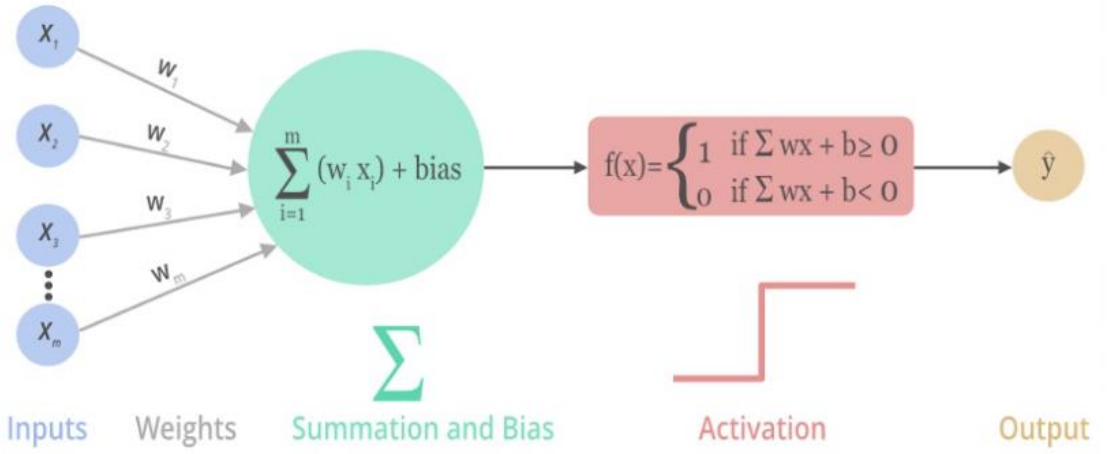



图 1.3 数学可视化神经元工作原理

一般来说，在神经元中，激活函数是很重要的一部分，为了增强网络的表示能力和学习能力，神经网络的激活函数都是非线性的，通常具有以下几点性质：

- 连续并可导（允许少数点上不可导），可导的激活函数可以直接利用数值优化的方法来学习网络参数；
- 激活函数及其导数要尽可能简单一些，太复杂不利于提高网络计算率；
- 激活函数的导函数值域要在一个合适的区间内，不能太大也不能太小，否则会影响训练的效率和稳定性。

| 激活函数 | 形式 | 导数形式 |
| --- | --- | --- |
| Sigmoid | $f(x)=\frac{1}{1+e^{-x}}$ | $f'(x)=f(x)(1-f(x))$ |
| Tanh | $f(x)=tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ | $f'(x)=1-(f(z))^2$ |
| ReLU | $f(x)=max(0,x)=\begin{cases}0 & x\leq 0\\ x & x>0\end{cases}$ | $f'(x)=\begin{cases}0 & x\leq 0\\ 1 & x>0\end{cases}$ |
| Leaky ReLU | $f(x)=max(0.001x,x)=\begin{cases}0.001x & x\leq 0\\ x & x>0\end{cases}a$ | $f(x)=max(0.001x,x)=\begin{cases}0.001 & x\leq 0\\ 1 & x>0\end{cases}$ |
| PReLU | $f(x)=max(\alpha x,x)=\begin{cases}\alpha x & x\leq 0\\ x & x>0\end{cases}$ | $f(x)=max(\alpha x,x)=\begin{cases}\alpha & x\leq 0\\ 1 & x>0\end{cases}$ |
| RReLU | PReLU中的 $\alpha$ 随机取值 | |
| ELU | $f(x)=\begin{cases}x & x\geq 0\\ \alpha(e^{x}-1) & x<0\end{cases}$ | $f(x)=\begin{cases}1 & x\geq 0\\ \alpha e^{x} & x<0\end{cases}$ |
| Maxout | $f(x)=max(w_1^Tx+b_1,w_2^Tx+b_2)$ | $f(x)=max(w_1,w_2)$ |

https://blog.csdn.net/weixin_39139473

### （1） Sigmoid 函数

Sigmoid 函数也叫 Logistic 函数，用于隐层神经元输出，取值范围为(0,1)，它可以将一个实数映射到(0,1)的区间，可以用来做二分类。在特征相差比较复杂或是相差不是特别大时效果比较好。sigmoid 是一个十分常见的激活函数，函数的表达式如下：

$$\boldsymbol{sigmoid(x) = \frac{1}{1 + exp\ (-x)}}$$

sigmoid 函数在早期的神经网络中较为普遍，但它目前逐渐被更简单的 ReLU 函数取代。下面绘制了 sigmoid 函数，如图 1.3 所示。当输入接近 0 时，sigmoid 函数接近线性变换。

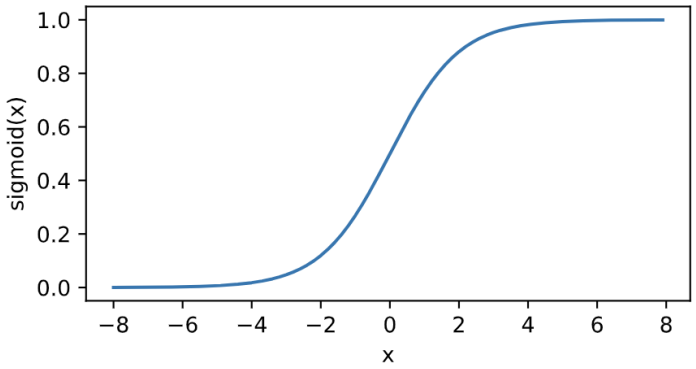



图 1.3 sigmoid 函数

在什么情况下适合使用 Sigmoid 激活函数呢？

- Sigmoid 函数的输出范围是 0 到 1。由于输出值限定在 0 到 1，因此它对每个神经元的输出进行了归一化；
- 用于将预测概率作为输出的模型。由于概率的取值范围是 0 到 1，因此 Sigmoid 函数非常合适；
- 梯度平滑，避免「跳跃」的输出值；
- 函数是可微的。这意味着可以找到任意两个点的 sigmoid 曲线的斜率；
- 明确的预测，即非常接近 1 或 0。

Sigmoid 激活函数存在的不足：

- 梯度消失：注意：Sigmoid 函数趋近 0 和 1 的时候变化率会变得平坦，也就是说，Sigmoid 的梯度趋近于 0。神经网络使用 Sigmoid 激活函数进行反向传播时，输出接近 0 或 1 的神经元其梯度趋近于 0。这些神经元叫作饱和神经元。因此，这些神经元的权重不会更新。此外，与此类神经元相连的神经元的权重也更新得很慢。该问题叫作梯度消失。因此，想象一下，如果一个大型神经网络包含 Sigmoid 神经元，而其中很多个都处于饱和状态，那么该网络无法执行反向传播。
- 不以零为中心：Sigmoid 输出不以零为中心的，，输出恒大于 0，非零中心化的输出会使得其后一层的神经元的输入发生偏置偏移（Bias Shift），并进一步使得梯度下降的收敛速度变慢。
- 计算成本高昂：exp() 函数与其他非线性激活函数相比，计算成本高昂，计算机运行起来速度较慢。

### （2） Tanh 函数

Tanh 激活函数又叫作双曲正切激活函数（hyperbolic tangent activation function）。其公式为：

$$\boldsymbol{tanh(x) = \frac{1 - exp(-2x)}{1 + exp(-2x)}}$$

tanh 是一个双曲正切函数。tanh 函数和 sigmoid 函数的曲线相对相似。但是它比 sigmoid 函数更有一些优势。首先，当输入较大或较小时，输出几乎是平滑的并且梯度较小，这不利于权重更新。二者的区别在于输出间隔，tanh 的输出间隔为 1，并且整个函数以 0 为中心，比 sigmoid 函数更

好。在 tanh 图中，负输入将被强映射为负，而零输入被映射为接近零。

注意：在一般的二元分类问题中，tanh 函数用于隐藏层，而 sigmoid 函数用于输出层，但这并不是固定的，需要根据特定问题进行调整。

Tanh 函数如图 1.4 所示。

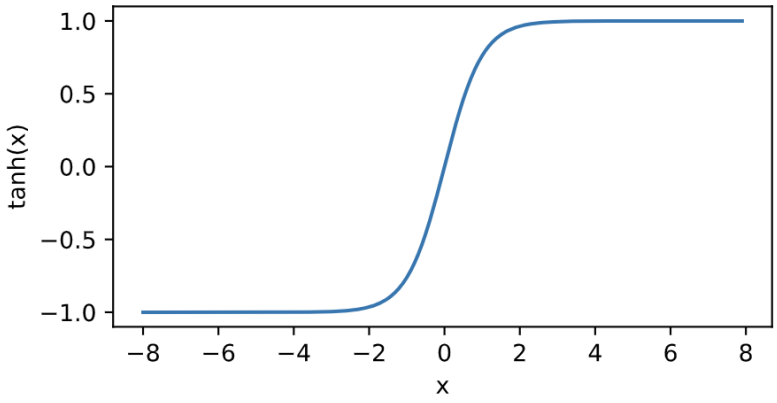



图 1.4 tanh 函数

**（3） ReLu(Rectified linear unit)函数**

ReLU 函数又称为修正线性单元（Rectified Linear Unit），是一种分段线性函数，其弥补了 sigmoid 函数以及 tanh 函数的梯度消失问题，在目前的深度神经网络中被广泛使用。ReLU 函数本质上是一个斜坡（ramp）函数，公式及函数图像如下：

$$ReLU(x) = max(x, 0)$$

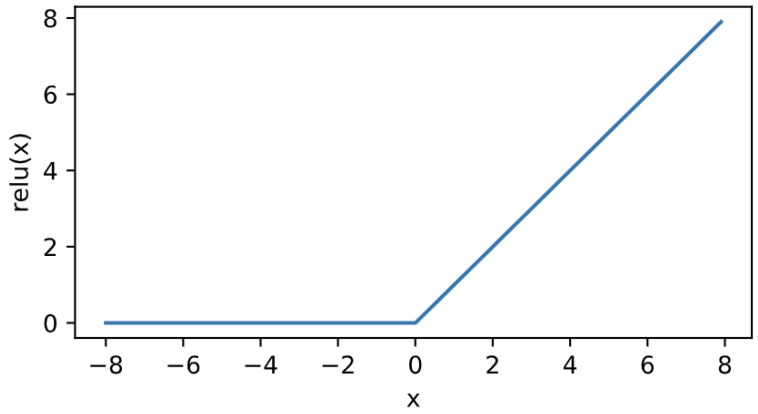



图 1.5 ReLu 函数

相比于 sigmoid 函数和 tanh 函数，当输入为正时，不存在梯度饱和问题，计算速度快得多。此外，当输入为负数时，ReLU 函数的导数为 0，完全失效(Dead ReLU)；当输入为正数时，ReLU 函数的导数为 1。

**（4） Leaky ReLu 函数**

$$LeayReLU(x) = \begin{cases} \alpha x, & if\ x < 0 \\ x, & if\ x \geq 0 \end{cases}$$

它是一种专门设计用于解决 Dead ReLU 问题的激活函数：

Leaky ReLU 通过把 x 的非常小的线性分量给予负输入（α）来调整负值的零梯度（zero gradients）问题，有助于扩大 ReLU 函数的范围，通常 α 的值为 0.01 左右。Leaky ReLU 的函数范围是（负无穷到正无穷）。

注意：从理论上讲，Leaky ReLU 具有 ReLU 的所有优点，而且 Dead ReLU 不会有任何问题，但在实际操作中，尚未完全证明 Leaky ReLU 总是比 ReLU 更好。

**（5） ELU(Exponential Linear Units)函数**

$$ELU(x) = \begin{cases} \alpha(e^x - 1), & if\ x < 0 \\ x, & if\ x \geq 0 \end{cases}$$

具有 relu 的优势，没有 Dead ReLU 问题，输出均值接近 0。有负数饱和区域，从而对噪声有一些鲁棒性。ELU 通过减少偏置偏移的影响，使**正常梯度**更接近于**单位自然梯度**，从而使**均值向零加速学习**可以看做是介于 ReLU 和 Leaky ReLU 之间的一个函数。当然，这个函数也需要计算 exp，从而计算量上更大一些。

**（6） PReLU(Parametric Rectified linear unit)函数**

$$PReLU(x) = \begin{cases} \alpha_i x_i, & if\ x < 0 \\ x_i, & if\ x \geq 0 \end{cases}$$

函数同 LeakyReLU，只是 α 是动态参数，通过反向传播学习的。在负值域，PReLU 的斜率较小，这也可以避免 Dead ReLU 问题。与 ELU 相比，PReLU 在负值域是线性运算。尽管斜率很小，但不会趋于 0。

**（7） GeLU(Gaussian error linear unit)函数**

ReLU 等分段线性函数并不光滑，且在某些间断点处不可导(特别是原点处)。然而，神经网络处理的数据一般要求是零均值的，ReLU 等函数在零点不可微，会在某种程度上影响网络的性能。此外，ReLU 等确定性非线性函数作为神经网络的激活函数时，网络往往需要加入随机正则以提高模型的泛化性能。如果一个非线性函数本身具有随机正则性，就能同时保证网络的非线性性和泛化性。

高斯误差线性单元(Gaussian error linear unit, GELU) 使用一个依赖于输入本身的概率统计量为激活函数提供随机正则性，同时保持输入信息。

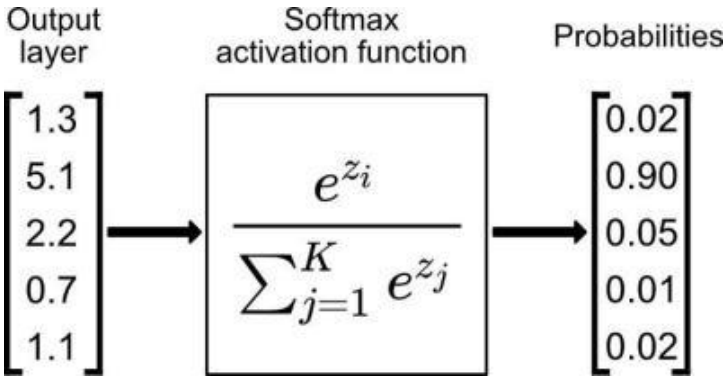
$$\mathrm{GELU}(x) = x * P(X \leq x) = x * \Phi(x) = x \int_{-\infty}^{x} \frac{e^{-\frac{(X-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma} dX$$

其中$\Phi(x)$指的是 x 的高斯正态分布的累积函数，$\mu$和$\sigma$分别代表正太分布的均值和标准差。

将 p 作为伯努利分布的概率参数,随机采样出 0 或 1，0 则输出被 mask，否则不被 mask。和 relu 不同的地方在于，relu 对于小于 0 的输出是置 0 的，而 gelu 则是有一定的概率不置 0

**（8） Softmax 函数**

$$\mathrm{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$




Softmax 是用于多类分类问题的激活函数，在多类分类问题中，超过两个类标签则需要类成员关系。对于长度为 K 的任意实向量，Softmax 可以将其压缩为长度为 K，值在（0，1）范围内，并且向量中元素的总和为 1 的实向量。

Softmax 函数的分母结合了原始输出值的所有因子，这意味着 Softmax 函数获得的各种概率彼此相关。

Softmax 激活函数的主要缺点是：1、在零点不可微；2、负输入的梯度为零，这意味着对于该区域的激活，权重不会在反向传播期间更新，因此会产生永不激活的死亡神经元。

**（9） 激活函数可视化**

参考：https://dashee87.github.io/deep%20learning/visualising-activation-functions-in-neural-networks/

### 1.1.3 多层感知机(MLP)

多层感知机就是含有至少一个隐藏层的由全连接层组成的神经网络，且每个隐藏层的输出通过激活函数进行变换。多层感知机的层数和各隐藏层中隐藏单元个数都是超参数。以单隐藏层为例并沿用本节之前定义的符号，多层感知机按以下方式计算输出：

$$H = \phi(XW_h + b_h)$$

$$O = HW_o + b_o$$

其中 ϕ 表示激活函数。在分类问题中，我们可以对输出$O$做 softmax 运算，并使用 softmax 回归中的交叉熵损失函数。在回归问题中，我们将输出层的输出个数设为 1，并将输出$O$直接提供给线性回归中使用的平方损失函数。

## 1.2 深度神经网络（Deep Neural Networks）

深度神经网络 (DNN) 只是一个具有深层的人工神经网络。在这种情况下，深层意味着网络有几个层堆叠在一起用于处理和从数据中学习。DNN 最初受到神经生物学的启发。特别是人类学习感知和识别物理事物的方式。请务必注意，MLP 被视为 DNN 的示例。DNN 的其他示例可以在卷积神经网络(CNN) 和递归神经网络(RNN)中找到。由于其复杂性，DNN 通常需要很长时间才能根据输入数据训练网络。

与 MLP 的结构类似，DNN 由输入层、隐藏层、输出层、权重、偏差和激活函数组成。或者，在 CNN 的情况下，除了已经提到的组件之外，神经网络将由池化层和卷积层组成。

DNN 因其深层而成为强大的算法。因此，它们通常用于处理复杂的计算任务。计算机视觉就是其中之一。这是人工智能的一个分支，旨在使计算机能够从图像和视频等视觉数据中获取意义。计算机视觉任务是使用 CNN 完成的。CNN 接收图片和视频形式的输入数据，然后处理这些数据。处理以计算机能够识别类似格式的图像的方式完成。这个过程类似于人类学习观察和感知周围环境的方式。DNN 的其他应用示例包括自然语言处理、基因组学、机器翻译、语音识别和自动驾驶汽车。

## 1.3 正向传播、反向传播(back propagation)

### 1.3.1 正向传播

正向传播是指对神经网络沿着从输入层到输出层的顺序，依次计算并存储模型的中间变量（包括输出）。为简单起见，假设输入是一个特征为$x \in R^d$的样本，且不考虑偏差项，那么中间变量为:

$$Z = W^{(1)}x,$$

其中$W^{(1)} \in R^{h \times d}$是隐藏层的权重参数。把中间变量$z \in R^h$输入按元素运算的激活函数 ϕ 后，将得到向量长度为 h 的隐藏层变量:

$$h = \emptyset(z)$$

隐藏层变量 h 也是一个中间变量。假设输出层参数只有权重$W^{(2)} \in R^{q \times h}$，可以得到向量长度为 q 的输出层变量

$$o = W^{(2)}h$$

假设损失函数为$l$，且样本标签为 y，可以计算出单个数据样本的损失项:

$$L = l(o, y)$$

根据 L2 范数正则化的定义，给定超参数 λ，正则化项即

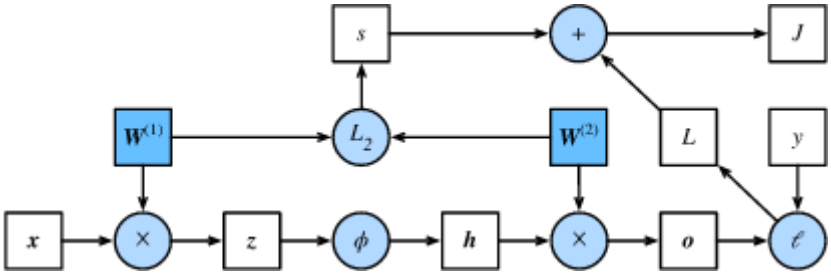
$$s = \frac{\lambda}{2}(\| W^{(1)} \|_F^2 + \| W^{(2)} \|_F^2)$$

其中$\lambda$是正则项系数，矩阵的 Frobenius 范数等价于将矩阵变平为向量后计算 L2 范数。最终，模型在给定的数据样本上带正则化的损失为:

$$J = L + s.$$

我们将 J 称为有关给定数据样本的目标函数，并在以下的讨论中简称目标函数。

我们通常绘制计算图来可视化运算符和变量在计算中的依赖关系。图 3.6 绘制了本节中样例模型正向传播的计算图，其中左下角是输入，右上角是输出。可以看到，图中箭头方向大多是向右和向上，其中方框代表变量，圆圈代表运算符，箭头表示从输入到输出之间的依赖关系。




## 1.3.2 反向传播

反向传播是“误差反向传播”的缩写，是一种使用梯度下降对人工神经网络进行监督学习的算法。给定人工神经网络和误差函数，该方法计算误差函数相对于神经网络权重的梯度。它是感知器对多层前馈神经网络的 delta 规则的推广。名称中的“向后”部分源于这样一个事实，即梯度的计算通过网络向后进行，首先计算最后一层权重的梯度，最后计算第一层权重的梯度。来自一层的梯度的部分计算在前一层的梯度计算中被重复使用。这种错误信息的反向流动允许有效地计算每一层的梯度，而不是单独计算每一层梯度的朴素方法。鉴于深度神经网络在图像识别和语音识别中的广泛采用，反向传播最近重新流行起来。

反向传播指的是计算神经网络参数梯度的方法。总的来说，反向传播依据微积分中的链式法则，沿着从输出层到输入层的顺序，依次计算并存储目标函数有关神经网络各层的中间变量以及参数的梯度。对输入或输出 X,Y,Z 为任意形状张量的函数 Y=f(X)和 Z=g(Y)，通过链式法则，有：

$$\frac{\partial Z}{\partial x} = prod(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X})$$

其中 prod 运算符将根据两个输入的形状，在必要的操作（如转置和互换输入位置）后对两个输入做乘法。

反向传播的目标是计算$\partial \mathrm{J}/\partial \mathrm{W}^{(1)}$ 和 $\partial \mathrm{J}/\partial \mathrm{W}^{(2)}$。我们将应用链式法则依次计算各中间变量和参数的梯度，其计算次序与前向传播中相应中间变量的计算次序恰恰相反。首先，分别计算目标函数 J=L+s 有关损失项 L 和正则项 s 的梯度:

$$\frac{\partial J}{\partial L} = 1, \frac{\partial J}{\partial S} = 1$$

其次，依据链式法则计算目标函数有关输出层变量的梯度$\frac{\partial J}{\partial o} \in R^q$:

$$\frac{\partial J}{\partial o} = prod\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial o}\right) = \frac{\partial L}{\partial o}$$

接下来，计算正则项有关两个参数的梯度：

$$\frac{\partial S}{\partial W^{(1)}} = \lambda W^{(1)}, \quad \frac{\partial S}{\partial W^{(2)}} = \lambda W^{(2)}$$

现在，我们可以计算最靠近输出层的模型参数的梯度$\partial J/\partial W^{(2)} \in R^{q \times h}$。依据链式法则，得到：

$$\frac{\partial J}{\partial W^{(2)}} = prod\left(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial W^{(2)}}\right) + prod\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(2)}}\right) = \frac{\partial J}{\partial o} h^T + \lambda W^{(2)}$$

沿着输出层向隐藏层继续反向传播，隐藏层变量的梯度$\partial J/\partial h \in R^q$可以这样计算：

$$\frac{\partial J}{\partial h} = prod\left(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial h}\right) = W^{(2)^T} \frac{\partial J}{\partial o}$$

由于激活函数 ϕ 是按元素运算的，中间变量$Z$的梯度$\partial J/\partial Z \in R^{h\times d}$。依据链式法则，得到：

$$\frac{\partial J}{\partial Z} = prod\left(\frac{\partial J}{\partial h}, \frac{\partial h}{\partial Z}\right) = \frac{\partial J}{\partial h} \odot \Phi'(Z)$$

最终，我们可以得到最靠近输入层的模型参数的梯度$\partial J/\partial W^{(1)} \in R^{h\times d}$。依据链式法则，得到：

$$\frac{\partial J}{\partial W^{(1)}} = prod\left(\frac{\partial J}{\partial Z}, \frac{\partial Z}{\partial W^{(1)}}\right) + prod\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(1)}}\right) = \frac{\partial J}{\partial Z} x^T + \lambda W^{(1)}$$

## 1.4 优化方法

机器学习的主要目标是创建一个性能良好的模型，并在特定情况下给出准确的预测。为了实现这一目标，我们需要机器学习优化。具有动量的随机梯度下降、RMSProp 和 Adam Optimizer 是专门为深度学习优化而创建的算法。

### 1.4.1 梯度下降（Gradient descent）

梯度下降是基于遵循场景局部坡度的基本思想。如图 1.6。微积分为我们提供了一种计算场景坡度的优化方法，坡度是函数在该点（也称为梯度）相对于权重的导数。其公式为：

$$w = w - learning\_rate \cdot \nabla wL(w)$$

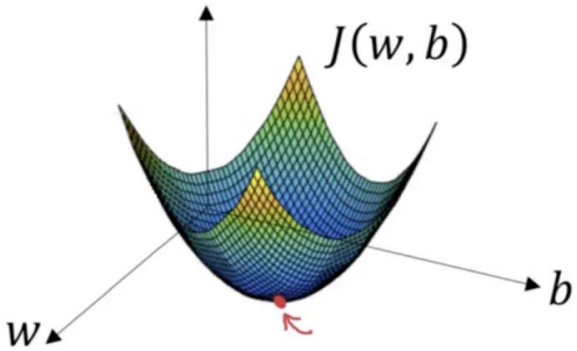
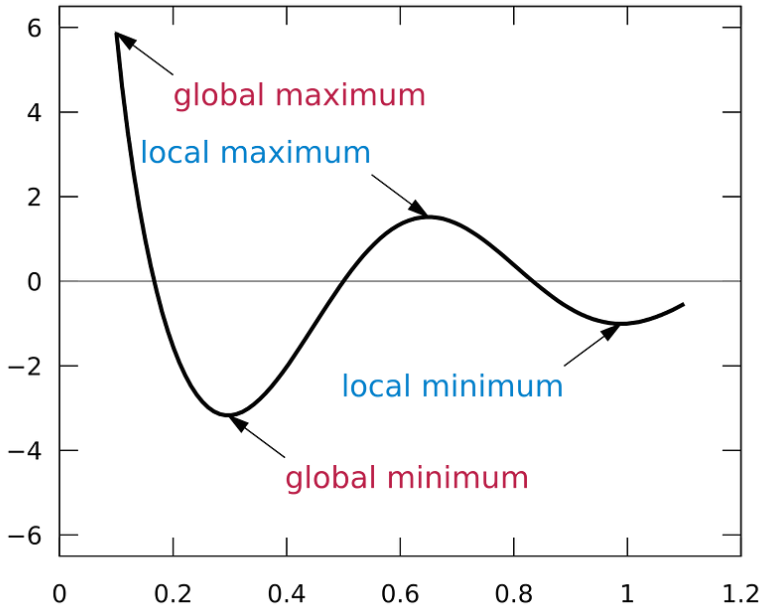



图 1.6 梯度下降算法

至于梯度下降算法，一次加载整个数据集。这使得它的计算量很大。另一个缺点是迭代值有可能卡在局部最小值或鞍点并且永远不会收敛到最小值，如图 1.7 所示。为了获得最佳解决方案，必须达到全局最小值。



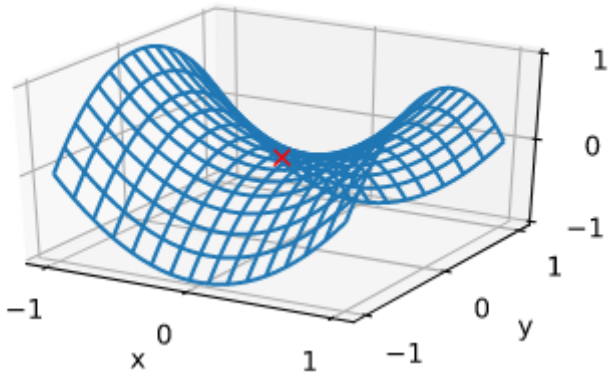




图 1.7 梯度下降算法示意图 1

从算法上讲，这可以用 Python 表示如下：

```
for t in range(steps):
  dw = gradient(loss, data, w)
  w = w - learning_rate *dw
```

在视觉上，我们可以想象下图对应一个二维空间。

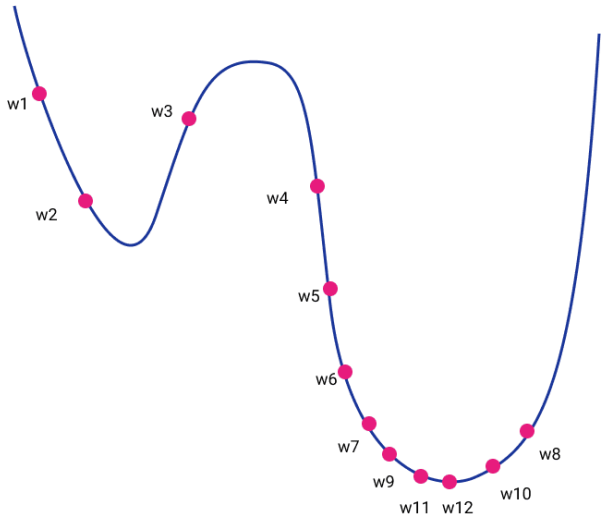



图 1.8 梯度下降算法示意图 2

实际上，在深度学习方面，梯度下降有 3 种主要的不同变体。

**（1） 批量梯度下降（Batch gradient descent）**

上面给出的等式和代码实际上指的是批量梯度下降。在这个变体中，我们在更新权重之前计算每个训练步骤中整个数据集的梯度。公式为：

$$\nabla wL(w) = \frac{1}{N}\sum_{i=1}^{N} \nabla w\, Li(xi, yi, W)$$

可以想象，由于我们对所有单个训练实例的损失求和，我们的计算很快就会变得非常耗时。因此对于大型数据集来说是不切实际的。

**（2） 随机梯度下降（Stochastic gradient descent）**

引入随机梯度下降（SGD）来解决这个确切的问题。SGD 不是计算所有训练实例的梯度并更新权重，而是更新每个训练实例的权重$xi, yi$,其公式为：

$$w = w - learning\_rate \cdot \nabla wL(xi, yi, W)$$

用 Python 表示为：

```
for t in range(steps):
  for example in data:
    dw = gradient(loss, example, w)
    w = w - learning_rate *dw
```

因此，SGD 速度更快，计算效率更高，但它在梯度估计中存在噪声。由于它频繁更新权重，因此会导致较大的振荡，从而使训练过程非常不稳定。你可以想象我们不断地沿着场景走锯齿形，这导致不断超调并错过我们的最小值,如下图 1.9。尽管出于同样的原因，我们可以轻松地摆脱局部最小值并继续寻找更好的值。

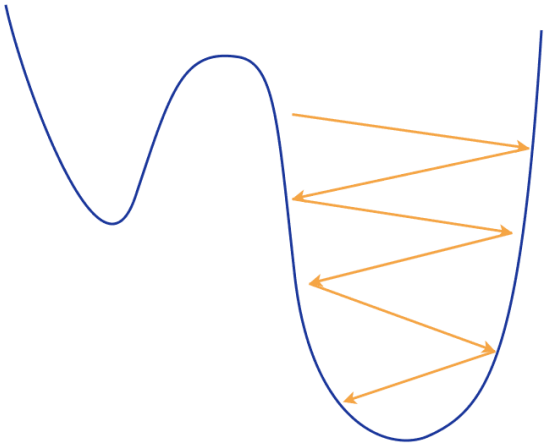


图 1.9 SGD

**（3） 小批量梯度下降（Mini-batch Stochastic Gradient Descent）**

小批量 SGD 结合了两个方法的优点。它随机选择训练样本，即所谓的小批量，来自整个数据集，并仅从中计算梯度。它本质上试图通过仅对数据的一个子集进行采样来近似批量梯度下降。公式为：

$$w = w - learning\_rate \cdot \nabla wL(x(i{:}i+n), y(i{:}i+n), W)$$

用 python 表示为：

```
for t in range(steps):

  for mini_batch in get_batches(data, batch_size):

    dw = gradient(loss, mini_batch, w)

    w = w - learning_rate *dw
```

在实践中，小批量 SGD 是最常用的变体，因为它的计算成本低，而且收敛性更强。

然而，这个基本版本的 SGD 有一些限制和问题，可能会对训练产生负面影响。

- 如果损失函数在一个方向上快速变化而在另一个方向上变化缓慢，则可能导致梯度的高度振荡，从而使训练进度非常缓慢。
- 如果损失函数有局部最小值或鞍点，SGD 很可能会卡在那里而无法“跳出”并继续寻找更好的最小值。发生这种情况是因为梯度变为零，因此权重没有任何更新。
- 梯度仍然有噪声，因为我们仅根据数据集的一小部分样本来估计它们。嘈杂的更新可能与损失函数的真实方向没有很好的关联。
- 选择一个好的损失函数是很棘手的，需要用不同的超参数进行耗时的实验。
- 相同的学习率应用于我们所有的参数，这对于具有不同频率或重要性的特征可能会产生问题。

为了克服这些问题中的一些，多年来已经提出了许多改进。

## 1.4.2 增加动力（Adding Momentum）

对 SGD 的一项基本改进来自添加动量的概念。借用物理学中的动量原理，我们强制 SGD 保持与之前时间步相同的方向移动。为此，我们引入了两个新变量：速度和摩擦力。

- 速度 v 被计算为直到该时间点的梯度的运行平均值，并指示梯度应继续朝哪个方向移动。
- 摩擦 ρ 是一个旨在衰减的常数。

在每个时间步，我们通过将先前的速度衰减一个因子来更新我们的速度 v，并且我们在当前时间添加权重的梯度。然后我们在速度向量的方向上更新我们的权重。

$$v_{t+1} = \rho v_t + \nabla wL(x, y, W)$$

$$w = w - learning\_rate \cdot v_{t+1}$$

**但是我们从动量中获得了什么？**

- 我们现在可以避开局部最小值或鞍点，因为即使小批量的梯度可能为零，我们也会继续向下移动
- 动量还可以帮助我们减少梯度的振荡，因为速度矢量可以平滑这些高度变化的场景。
- 最后，它减少了梯度的噪声（随机性）并沿着场景更直接地行走。

```
for t in range(steps):

    dw = gradient(loss, w)

    v = rho*v +dw

    w = w - learning_rate *v
```

## 1.4.3 自适应学习率（Adaptive Learning Rate）

优化算法的另一个重要思想是自适应学习率。直觉是我们希望对频繁的特征进行较小的更新，对不频繁的特征进行较大的更新。这将使我们能够克服之前提到的 SGD 的一些问题。

（1）Adagrad

Adagrad 保留每个维度中梯度平方的运行总和，并且在每次更新中，我们根据总和调整学习率。这样我们就可以为每个参数实现不同的学习率（或自适应学习率）。此外，通过使用平方梯度的根，我们只考虑梯度的大小而不是符号。

$$w = w - \frac{learning\_rate}{\sqrt{G} + e} \odot \nabla wL(x, y, W)$$

Where $G = \sum_{t=1}^{T} \nabla wL(x, y, w_t)^2$，表示参数的前 T 次梯度的平方累加和。

```
for t in range(steps):

    dw = gradient(loss, w)

    squared_gradients += dw*dw

    w = w - learning_rate * dw/ (squared_gradients.sqrt() + e)
```

我们可以看到，当梯度变化非常快时，学习率会变小。当梯度变化缓慢时，学习率会更大。Adagrad 的一大缺点是随着时间的推移，由于运行平方和的单调递增，学习率越来越小。

（2）RMSprop

这个问题的解决方案是对上述算法进行修改，称为 RMSProp，可以将其视为“Leaky Adagrad”。本质上，我们通过衰减之前的平方梯度之和再次添加了摩擦的概念。正如我们在基于动量的方法中所做的那样，我们将项（此处为运行平方和）乘以常数值（衰减率）。这样我们希望算法不会像 Adagrad 那样在训练过程中变慢。

$$v_t = \rho v_{t-1} + (1 - \rho)(\nabla wL(x, y, w_t))^2$$

$$w = w - \frac{learning\_rate}{\sqrt{v_t} + e} \odot \nabla wL(x, y, W)$$

```
for t in range(steps):

    dw = gradient(loss, w)

    squared_gradients = decay_rate*squared_gradients + (1- decay_rate)* dw*dw

    w = w - learning_rate * dw/(squared_gradients.sqrt() + e)
```

在大多数自适应速率算法中，添加了一个非常小的值 e 以防止分母无效。通常它等于 1e-7。

## 1.4.4 Adam

Adam（自适应矩估计）可以说是当今最流行的变体。它已广泛用于研究和商业应用。它的流行在于它结合了以前最好的两个方法。动量和自适应学习率。现在跟踪两个运行变量，速度和 RMSProp 上描述的平方梯度平均值。它们也被称为一阶矩和二阶矩。

$$m_t = \delta_1 m_{t-1} + (1 - \delta_1)\nabla wL(x, y, W)$$
$$v_t = \delta_2 v_{t-1} + (1 - \delta_2)\nabla wL(x, y, W)^2$$
$$\widehat{m_t} = \frac{m_t}{1 - {\delta_1}^t}$$
$$\widehat{v_t} = \frac{v_t}{1 - {\delta_2}^t}$$
$$w = w - \frac{learning\_rate}{\sqrt{\widehat{v_t}} + e}\widehat{m_t}$$

```
for t in range(steps):

    dw = gradient(loss, w)

    moment1= delta1 *moment1  +(1-delta1)* dw

    moment2 = delta2*moment2 +(1-delta2)*dw*dw

    moment1_unbiased = moment1  /(1-delta1**t)

    moment2_unbiased = moment2  /(1-delta2**t)

    w = w - learning_rate*moment1_unbiased/ (moment2_unbiased.sqrt()+e)
```

# 1.5 损失函数

损失函数用于衡量预测输出与提供的目标值之间的误差。
例如，损失函数（我们称之为 J）可以采用以下两个参数：

- 预测输出（ y_pred ）
- 目标值（ y ）

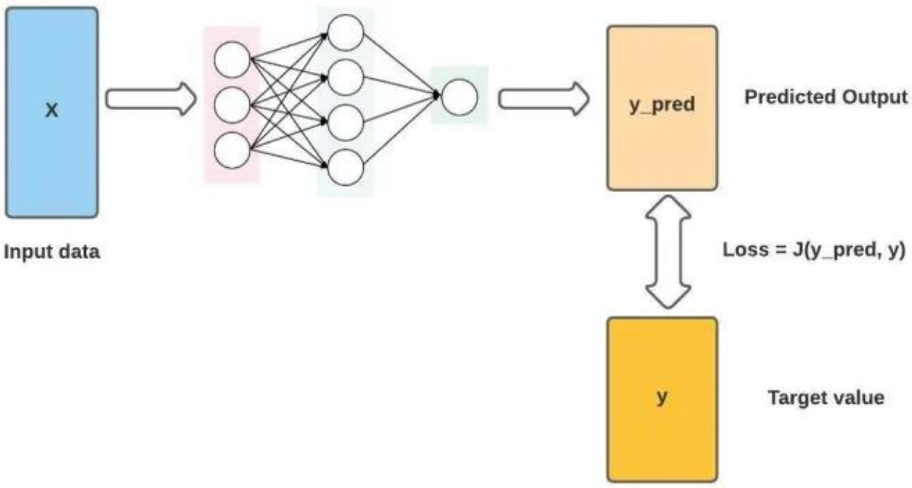



图 1.10 神经网络损失图

此函数将通过将预测输出与预期输出进行比较来确定模型的性能。如果 y_pred 和 y 的偏差很大，loss 值会很高。如果偏差很小或值几乎相同，它将输出非常低的损失值。因此，当模型在提供的数据集上训练时，需要使用可以适当惩罚模型的损失函数。

## 1.5.1 平均绝对误差（L1 Loss Function）

平均绝对误差(MAE)，也称为 L1 损失，计算实际值与预测值之间的绝对差之和的平均值它检查一组预测值中误差的大小，而不关心它们的正方向或负方向。如果不使用误差的绝对值，则负值可以抵消正值。

$$loss(p,y) = |p - y|$$

y 代表实际值，p 代表预测值。

## 1.5.2 均方误差损失函数(Mean Squared Error Loss Function)

均方误差 (MSE)也称为 L2 损失，计算实际值和预测值之间的平方差的平均值。无论实际值和预测值的符号如何，Pytorch MSE Loss 始终输出正结果。为了提高模型的准确性，您应该尝试减少 L2 Loss——一个完美的值是 0.0。 平方意味着较大的错误比较小的错误产生更大的错误。如果分类器偏离 100，则错误为 10,000。如果偏离 0.1，则误差为 0.01。这会惩罚犯大错误的模型并鼓励小错误。

$$loss(p,y) = (p - y)^2$$

y 代表实际值，p 代表预测值。

## 1.5.3 交叉熵损失函数（Cross-Entropy Loss Function）

此损失函数针对提供的一组事件或随机变量计算两个概率分布之间的差异。它用于计算出一个分数，该分数总结了预测值和实际值之间的平均差异。为了提高模型的准确性，您应该尝试最小化分数——交叉熵分数介于 0 和 1 之间，完美值为 0。其他损失函数，如平方损失，惩罚不正确的预测。交叉熵会因为非常自信和错误而受到很大惩罚。与不根据预测置信度进行惩罚的负对数似然损失不同，交叉熵会惩罚不正确但置信度高的预测，以及正确但置信度较低的预测。 交叉熵函数有多种变体，其中最常见的类型是二元交叉熵 (BCE)。BCE Loss 主要用于二元分类模型；也就是说，模型只有 2 个类。

$$loss = -\frac{1}{N}\sum_{i=1}^{N} y_i log(p_i) + (1-y_i) log(1-p_i)$$

其中 N 是样本数，y 代表实际值，p 代表预测值。

# 1.6 卷积神经网络

## 1.6.1 卷积层

卷积神经网络（convolutional neural network）是含有卷积层（convolutional layer）的神经网络。

### （1）二维卷积运算

虽然卷积层得名于卷积（convolution）运算，但我们通常在卷积层中使用更加直观的卷积运算。在二维卷积层中，一个二维输入数组和一个二维核（kernel）数组通过卷积运算输出一个二维数组。我们用一个具体例子来解释二维卷积运算的含义。如图 1.11 所示，输入是一个高和宽均为 3 的二维数组。我们将该数组的形状记为 3×3 或（3，3）。核数组的高和宽分别为 2。该数组在卷积计算中又称卷积核或过滤器（filter）。卷积核窗口（又称卷积窗口）的形状取决于卷积核的高和宽，即 2×2。图 1.11 中的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素：0×0+1×1+3×2+4×3=19。

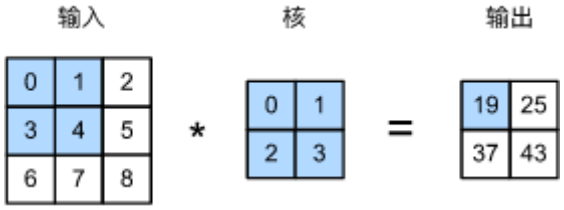



图 1.11 二维卷积运算

在二维卷积运算中，卷积窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。当卷积窗口滑动到某一位置时，窗口中的输入子数组与核数组按元素相乘并求和，得到输出数组中相应位置的元素。图 1.11 中的输出数组高和宽分别为 2，其中的 4 个元素由二维卷积运算得出：

0×0+1×1+3×2+4×3=19,
1×0+2×1+4×2+5×3=25,
3×0+4×1+6×2+7×3=37,
4×0+5×1+7×2+8×3=43.

### (2) 特征图和感受野

二维卷积层输出的二维数组可以看作是输入在空间维度（宽和高）上某一级的表征，也叫特征图（feature map）。影响元素 x 的前向计算的所有可能输入区域（可能大于输入的实际尺寸）叫做 x 的感受野（receptive field）。以图 1.11 为例，输入中阴影部分的四个元素是输出中阴影部分元素的感受野。我们将图 1.11 中形状为 2×2 的输出记为 Y，并考虑一个更深的卷积神经网络：将 Y 与另一个形状为 2×2 的核数组做卷积运算，输出单个元素 z。那么，z 在 Y 上的感受野包括 Y 的全部四个元素，在输入上的感受野包括其中全部 9 个元素。可见，我们可以通过更深的卷积神经网络使特征图中单个元素的感受野变得更加广阔，从而捕捉输入上更大尺寸的特征。

## 1.6.2 填充和步幅

一般来说，假设输入形状是$n_h \times n_w$，卷积核窗口形状是$k_h \times k_w$，那么输出形状将会是：

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

所以卷积层的输出形状由输入形状和卷积核窗口形状决定。此外，卷积层的两个超参数，即填充和步幅，它们可以基于给定形状的输入和卷积核改变输出形状。

### （1） 填充

填充（padding）是指在输入高和宽的两侧填充元素（通常是 0 元素）。在原输入高和宽的两侧分别添加了值为 0 的元素，使得输入高和宽从 3 变成了 5，并导致输出高和宽由 2 增加到 4。图 1.12 中的阴影部分为第一个输出元素及其计算所使用的输入和核数组元素：0×0+0×1+0×2+0×3=0。

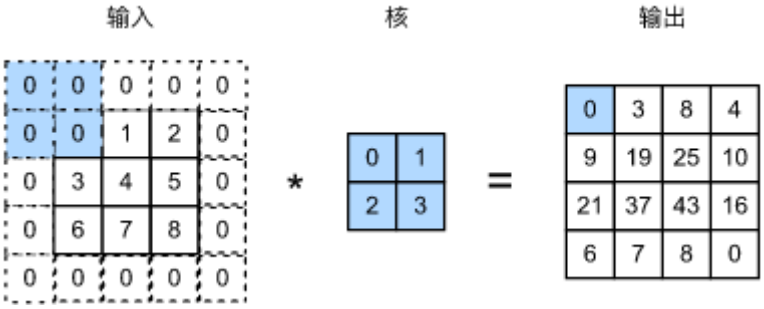



图 1.12 二维卷积计算

一般来说，如果在高的两侧一共填充$p_h$行，在宽的两侧一共填充$p_w$列，那么输出形状将会是：

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1).$$

也就是说，输出的高和宽会分别增加$p_h$和$p_w$。

在很多情况下，卷积神经网络经常使用奇数高宽的卷积核，如 1、3、5 和 7，设置$p_h = k_h - 1$和$p_w = k_w - 1$，两端上的填充个数相等来使输入和输出具有相同的高和宽，这样会方便在构造网络时推测每个层的输出形状。

### （2） 步幅

卷积窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。我们将每次滑动的行数和列数称为步幅（stride）。

默认在高和宽两个方向上步幅均为 1。我们也可以使用更大步幅。图 1.13 展示了在高上步幅为 3、在宽上步幅为 2 的二维卷积运算。可以看到，输出第一列第二个元素时，卷积窗口向下滑动了 3 行，而在输出第一行第二个元素时卷积窗口向右滑动了 2 列。当卷积窗口在输入上再向右滑动 2 列时，由于输入元素无法填满窗口，无结果输出。图 1.13 中的阴影部分为输出元素及其计算所使用的输入和核数组元素：0×0+0×1+1×2+2×3=8、0×0+6×1+0×2+0×3=6。

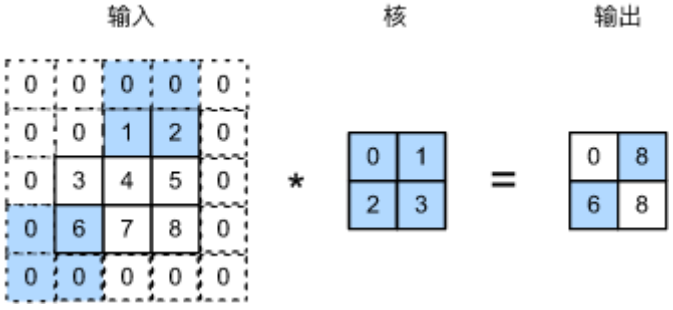



图 1.13 高和宽上步幅分别为 3 和 2 的二维卷积运算

一般来说，当高上步幅为$s_h$，宽上步幅为时$s_w$，输出形状为:

$$[(n_h - k_h + p_h + 1)/s_h] \times [(n_w - k_w + p_w + 1)/s_w]$$

## 1.6.3 多输入通道和多输出通道

彩色图像在高和宽 2 个维度外还有 RGB（红、绿、蓝）3 个颜色通道。假设彩色图像的高和宽分别是 h 和 w（像素），那么它可以表示为一个 3×h×w 的多维数组。我们将大小为 3 的这一维称为通道（channel）维。

### （1）多输入通道

当输入数据含多个通道时，我们需要构造一个输入通道数与输入数据的通道数相同的卷积核，从而能够与含多通道的输入数据做卷积运算。假设输入数据的通道数为$c_i$，那么卷积核的输入通道数同样为$c_i$。设卷积核窗口形状为$k_h \times k_w$。当$c_i = 1$时，我们知道卷积核只包含一个形状为$k_h \times k_w$的二维数组。当$c_i > 1$时，我们将会为每个输入通道各分配一个形状为$k_h \times k_w$ 的核数组。把这$c_i$个数组在输入通道维上连结，即得到一个形状为$c_i \times k_h \times k_w$的卷积核。由于输入和卷积核各有$c_i$个通道，我们可以在各个通道上对输入的二维数组和卷积核的二维核数组做卷积运算，再将这$c_i$个卷积运算的二维输出按通道相加，得到一个二维数组。这就是含多个通道的输入数据与多输入通道的卷积核做二维卷积运算的输出。

图 1.14 展示了含 2 个输入通道的二维卷积计算的例子。在每个通道上，二维输入数组与二维核数组做卷积运算，再按通道相加即得到输出。图 1.14 中阴影部分为第一个输出元素及其计算所使用的输入和核数组元素：(1×1+2×2+4×3+5×4)+(0×0+1×1+3×2+4×3)=56。

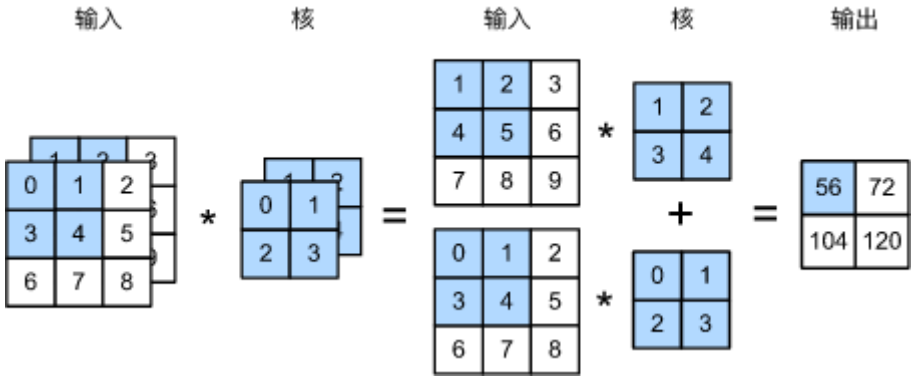



图 1.14 含 2 个输入通道的卷积计算

### （2）多输出通道

当输入通道有多个时，因为我们对各个通道的结果做了累加，所以不论输入通道数是多少，输出通道数总是为 1。设卷积核输入通道数和输出通道数分别为$c_i$和$c_o$，高和宽分别为$k_h$ 和$k_w$。如果希望得到含多个通道的输出，我们可以为每个输出通道分别创建形状为$c_i \times k_h \times k_w$的核数组。将它们在输出通道维上连结，卷积核的形状即$c_o \times c_i \times k_h \times k_w$。在做卷积运算时，每个输出通道上的结果由卷积核在该输出通道上的核数组与整个输入数组计算而来。

### （3）1x1 卷积层

一般来说，提升网络性能最直接的办法就是增加网络深度和宽度，这也就意味着巨量的参数。但是，巨量参数容易产生过拟合也会大大增加计算量。增加网络深度和宽度的同时减少参数，Inception 就是在这样的情况下应运

而生。

Inception 结构的主要思路是怎样用密集成分来近似最优的局部稀疏结构，基本结构如下图 1.15:

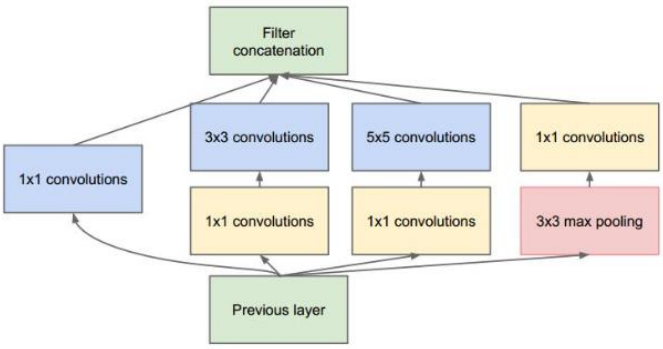



图 1.15 维度降低的 Inception 模块

采用不同大小的卷积核意味着不同大小的感受野，最后拼接意味着不同尺度特征的融合

之所以卷积核采用 1×1，3×3 和 5×5，主要是为了方便对齐，设定卷积步长 stride=1，只要分别设定 padding=0，1，2，那么卷积之后便可以得到相同维度的特征，然后将这些特征就可以直接拼接在一起了。

但是使用 5×5 的卷积核仍然会带来巨大的计算量，为此，采用 1×1 的卷积核来进行降维。

例如：上一层的输出为 100×100×128，经过具有 256 个输出的 5×5 卷积层之后 (stride=1，pad=2)，输出数据为 100×100×256。其中，卷积层的参数为 128×5×5×256。假如上一层输出先经过具有 32 个输出的 1×1 卷积层，再经过具有 256 个输出的 5×5 卷积层，那么最终的输出数据仍为为 100×100×256，但卷积参数量已经减少为 128×1×1×32 + 32×5×5×256，大约减少了 4 倍。

1×1 卷积层被当作保持高和宽维度形状不变的全连接层使用。可以通过调整网络层之间的通道数来控制模型复杂度。图 1.16 展示了使用输入通道数为 3、输出通道数为 2 的 1×1 卷积核的卷积计算。值得注意的是，输入和输出具有相同的高和宽。输出中的每个元素来自输入中在高和宽上相同位置的元素在不同通道之间的按权重累加。假设我们将通道维当作特征维，将高和宽维度上的元素当成数据样本，那么 1×1 卷积层的作用与全连接层等价。

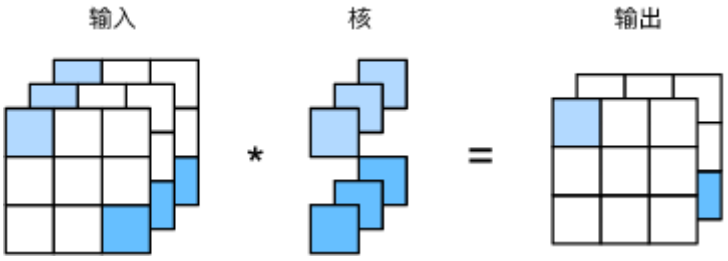



图 1.16 1x1 卷积核的卷积计算。输入和输出具有相同的高和宽

## 1.6.4 池化层

池化（pooling）层，它的提出是为了缓解卷积层对位置的过度敏感性。

### （1）二维最大池化层和平均池化层

同卷积层一样，池化层每次对输入数据的一个固定形状窗口（又称池化窗口）中的元素计算输出。不同于卷积层里计算输入和核的卷积，池化层直接计算池化窗口内元素的最大值或者平均值。该运算也分别叫做最大池化或平均池化。在二维最大池化中，池化窗口从输入数组的最左上方开始，按从左往右、从上往下的顺序，依次在输入数组上滑动。当池化窗口滑动到某一位置时，窗口中的输入子数组的最大值即输出数组中相应位置的元素。

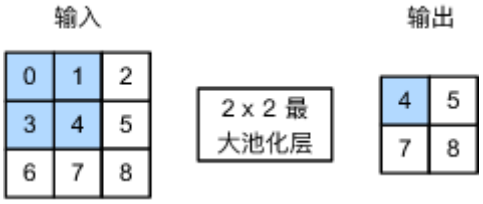



图 1.17 池化窗口形状为 2 x 2 的最大池化

图 1.17 展示了池化窗口形状为 2×2 的最大池化，阴影部分为第一个输出元素及其计算所使用的输入元素。输出数组的高和宽分别为 2，其中的 4 个元素由取最大值运算 max 得出：

$$\max(0,1,3,4)=4,$$
$$\max(1,2,4,5)=5,$$
$$\max(3,4,6,7)=7,$$
$$\max(4,5,7,8)=8$$

二维平均池化的工作原理与二维最大池化类似，但将最大运算符替换成平均运算符。池化窗口形状为 p×q 的池化层称为 p×q 池化层，其中的池化运算叫作 p×q 池化。

### （2）填充和步幅

同卷积层一样，池化层也可以在输入的高和宽两侧的填充并调整窗口的移动步幅来改变输出形状。池化层填充和步幅与卷积层填充和步幅的工作机制一样。

### （3）多通道

在处理多通道输入数据时，池化层对每个输入通道分别池化，而不是像卷积层那样将各通道的输入按通道相加。这意味着池化层的输出通道数与输入通道数相等。

## 1.6.5 卷积神经网络（LeNet）

卷积神经网络就是含卷积层的网络。本节里我们将介绍一个早期用来识别手写数字图像的卷积神经网络：LeNet。这个名字来源于 LeNet 论文的第一作者 Yann LeCun。LeNet 展示了通过梯度下降训练卷积神经网络可以达到手写数字识别在当时最先进的结果。这个奠基性的工作第一次将卷积神经网络推上舞台，为世人所知。LeNet 的网络结构如下图 1.18 所示。

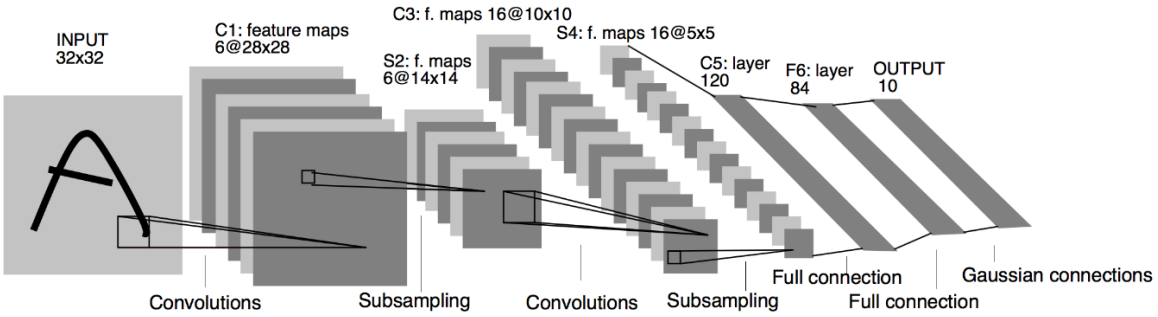



图 1.18 LeNet 网络结构

LeNet 分为卷积层块和全连接层块两个部分。下面我们分别介绍这两个模块。

卷积层块里的基本单位是卷积层后接最大池化层：卷积层用来识别图像里的空间模式，如线条和物体局部，之后的最大池化层则用来降低卷积层对位置的敏感性。卷积层块由两个这样的基本单位重复堆叠构成。在卷积层块中，每个卷积层都使用 5×5

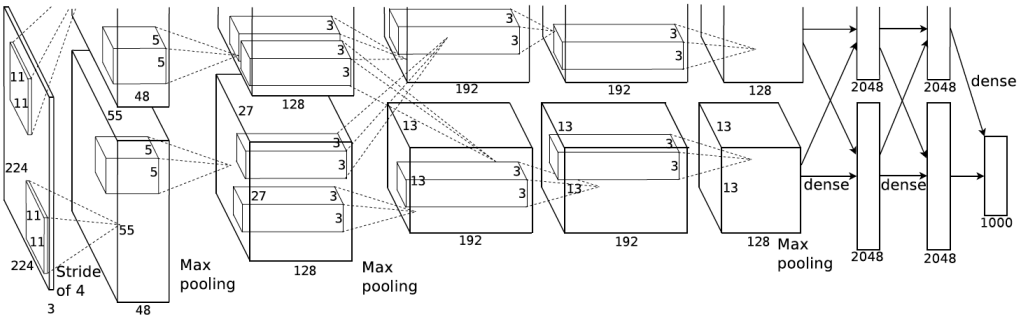
5×5 的窗口，并在输出上使用 sigmoid 激活函数。第一个卷积层输出通道数为 6，第二个卷积层输出通道数则增加到 16。这是因为第二个卷积层比第一个卷积层的输入的高和宽要小，所以增加输出通

道使两个卷积层的参数尺寸类似。卷积层块的两个最大池化层的窗口形状均为 2×2，且步幅为 2。由于池化窗口与步幅形状相同，池化窗口在输入上每次滑动所覆盖的区域互不重叠。

卷积层块的输出形状为(批量大小，通道，高，宽)。当卷积层块的输出传入全连接层块时，全连接层块会将小批量中每个样本变平（flatten)。也就是说，全连接层的输入形状将变成二维，其中第一维是小批量中的样本，第二维是每个样本变平后的向量表示，且向量长度为通道、高和宽的乘积。全连接层块含 3 个全连接层。它们的输出个数分别是 120、84 和 10，其中 10 为输出的类别个数。

## 1.6.6 深度卷积神经网络（AlexNet）

2012 年，AlexNet 横空出世。这个模型的名字来源于论文第一作者的姓名 Alex Krizhevsky [1]。AlexNet 使用了 8 层卷积神经网络，并以很大的优势赢得了 ImageNet 2012 图像识别挑战赛。它首次证明了学习到的特征可以超越手工设计的特征，从而一举打破计算机视觉研究的前状。




AlexNet 与 LeNet 的设计理念非常相似，但也有显著的区别。

第一，与相对较小的 LeNet 相比，AlexNet 包含 8 层变换，其中有 5 层卷积和 2 层全连接隐藏层，以及 1 个全连接输出层。下面我们来详细描述这些层的设计。

AlexNet 第一层中的卷积窗口形状是 11×11。因为 ImageNet 中绝大多数图像的高和宽均比 MNIST 图像的高和宽大 10 倍以上，ImageNet 图像的物体占用更多的像素，所以需要更大的卷积窗口来捕获物体。第二层中的卷积窗口形状减小到 5×5，之后全采用 3×3。此外，第一、第二和第五个卷积层之后都使用了窗口形状为 3×3、步幅为 2 的最大池化层。而且，AlexNet 使用的卷积通道数也大于 LeNet 中的卷积通道数数十倍。

紧接着最后一个卷积层的是两个输出个数为 4096 的全连接层。这两个巨大的全连接层带来将近 1 GB 的模型参数。由于早期显存的限制，最早的 AlexNet 使用双数据流的设计使一个 GPU 只需要处理一半模型。幸运的是，显存在过去几年得到了长足的发展，因此通常我们不再需要这样的特别设计了。

第二，AlexNet 将 sigmoid 激活函数改成了更加简单的 ReLU 激活函数。一方面，ReLU 激活函数的计算更简单，例如它并没有 sigmoid 激活函数中的求幂运算。另一方面，ReLU 激活函数在不同的参数初始化方法下使模型更容易训练。这是由于当 sigmoid 激活函数输出极接近 0 或 1 时，这些区域的梯度几乎为 0，从而造成反向传播无法继续更新部分模型参数；而 ReLU 激活函数在正区间的梯度恒为 1。因此，若模型参数初始化不当，sigmoid 函数可能在正区间得到几乎为 0 的梯度，从而令模型无法得到有效训练。

第三，AlexNet 通过丢弃法来控制全连接层的模型复杂度。而 LeNet 并没有使用丢弃法。

第四，AlexNet 引入了大量的图像增广，如翻转、裁剪和颜色变化，从而进一步扩大数据集来缓解过拟合。

参考：https://blog.csdn.net/jiaoyangwm/article/details/80011656

# 2.深度学习药物发现案例（一）

——药物-药物相互作用预测模型